

Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping

Chi-Keung Luk

SSG Software Pathfinding and Innovation
Intel Corporation, Hudson, MA
Email: chi-keung.luk@intel.com

Sunpyo Hong Hyesoon Kim

School of Computer Science
Georgia Institute of Technology, Atlanta, GA
Email: hyesoon@cc.gatech.edu

Abstract

Heterogeneous multiprocessors are growingly important in the multi-core era due to their potential for high performance and energy efficiency. In order for software to fully realize this potential, the step that maps computations to processing elements must be as automated as possible. However, the state-of-the-art approach is to rely on the programmer to specify this mapping manually and statically. This approach is not only labor intensive but also not adaptable to changes in runtime environments like problem sizes and hardware configurations. In this study, we propose *adaptive mapping*, a fully automatic technique to map computations to processing elements on heterogeneous multiprocessors. We have implemented it in our experimental heterogeneous programming system called *Qilin*. Our results demonstrate that, for a set of important computation kernels, automatic adaptive mapping achieves a speedup of 9.3x on average over the best serial implementation by judiciously distributing works over the CPU and GPU, which is 69% and 33% faster than using the CPU or GPU alone, respectively. In addition, adaptive mapping is within 94% of the speedup of the best manual mapping found via exhaustive searching. To the best of our knowledge, Qilin is the first and only system to date that has such capability.

1. Introduction

Multiprocessors have emerged as mainstream computing platforms nowadays. Among them, an increasingly popular class are those with *heterogeneous architectures*. By providing processing elements¹ of different performance/energy characteristics on the same machine, these architectures could deliver high performance and energy efficiency [10]. The most well-known heterogeneous architecture today is probably the IBM/Sony Cell architecture, which consists of a Power processor and eight synergistic processors [20]. In the personal computer (PC) world, a desktop now has a multicore CPU and a GPU, exposing multiple levels of hardware parallelism to software, as illustrated in Figure 1.

¹The term *processing element* (PE) is a generic term for a hardware element that executes a stream of instructions.

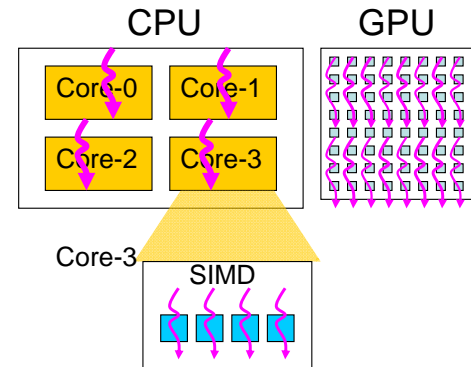


Figure 1. Multiple levels of hardware parallelism exposed to software on current CPU+GPU systems (The GPU has tens to hundreds of special-purpose cores, while the CPU has a few general-purpose cores. Within each CPU core, there is short-vector parallelism provided by the SIMD extension of the ISA.)

In order for mainstream programmers to fully tap into the potential of heterogeneous multiprocessors, the step that maps computations to processing elements must be as *automated* as possible. Unfortunately, the state-of-the-art approach [12, 17, 30] is to rely on the programmer to perform this mapping *manually*: for the Cell, O’Brien *et al.* extend the IBM XL compiler to support OpenMP on this architecture [17]; for commodity PCs, Linderman *et al.* propose the Merge framework [12], a library-based system to program CPU and GPU together using the map-reduce paradigm. In both cases, the computation-to-processor mappings are *statically* determined by the programmer. This manual approach not only puts burdens on the programmer but also is *not adaptable*, since the optimal mapping is likely to change with different applications, different input problem sizes, and different hardware configurations.

In this paper, we address this problem by introducing a *fully automatic* approach that decides the mapping from computations to processing elements using *run-time adaptation*. We have implemented our approach into *Qilin*², our experimental system for programming heterogeneous multiprocessors. Experimental results demonstrate that our automated adaptive mapping performs nearly as well as manual mapping while at the same time can tolerate changes in input problem sizes and hardware configurations. While Qilin currently focuses on CPU+GPU platforms, our adaptive map-

²“Qilin” is a mythical chimerical creature in the Chinese culture; we picked it as the project name for the heterogeneity of this creature.

ping technique is applicable to heterogeneous platforms in general, including the Cell and others, since our technique does *not* require any information regarding the hardware or machine models. To the best of our knowledge, Qilin is the first and the only system to date that has such capability.

The rest of this paper is organized as follows. In Section 2, we use a case study to further motivate the need of adaptive mapping. Section 3 then describes the Qilin system in details. We will focus on our runtime adaptation techniques in Section 4. Experimental evidences will be given in Section 5. Finally, we relate our works to others in Section 6 and conclude in Section 7.

2. A Case Study: Why Do We Need Adaptive Mapping?

We now motivate the need of adaptive mapping with a case study on matrix multiplication, a very commonly used computation kernel in scientific computing. We measured the parallelization speedups on matrix multiplication with a heterogeneous machine consisting of an Intel multicore CPU and a Nvidia multicore GPU (details of the machine configuration are given in Section 5.1).

We did three experiments with different input matrix sizes and the number of CPU cores used. In each experiment, we varied the distribution of work between the CPU and GPU. For matrix multiplication $C = A * B$, we first divide A into two smaller matrices A_1 and A_2 by rows. We then compute $C_1 = A_1 * B$ on the CPU and $C_2 = A_2 * B$ on the GPU in *parallel*. Finally, we obtain C by combining C_1 and C_2 . We use the best matrix multiplication libraries available: for the CPU, we use the Intel Math Kernel Library (MKL) [8]; for the GPU, we use the CUDA CUBLAS library [15].

Figure 2 shows the results of these three experiments. All input matrices are $N * N$ square matrices. The y-axis is the speedup over the serial case. The x-axis is the distribution of work across the CPU and GPU, where the notation “X/Y” means X% of work mapped to the GPU and Y% of work mapped to the CPU. At the two extremes are the cases where we schedule all the work on either the GPU or CPU.

In Experiment 1, we use a relatively small problem size ($N = 1000$) with eight CPU cores. The low computation-to-communication ratio with this problem size renders the GPU less effective. As a result, the optimal mapping is to schedule all work on the CPU. In Experiment 2, we increase the problem size to $N = 6000$ and keep the same number of CPU cores. Now, with a higher computation-to-communication ratio, the GPU becomes more effective—both the GPU-alone and the CPU-alone speedups are over 7x. And the optimal mapping is to schedule 60% of work on the GPU and 40% on the CPU, resulting into a 10.3x speedup. In Experiment 3, we keep the problem size to $N = 6000$ but reduce the number of CPU cores to only two. With much less CPU horsepower, the CPU-only speedup is limited to 2x and the optimal mapping now is to schedule 80% of work on the GPU and 20% on the CPU.

It is clear from these experiments that even for a single application, the optimal mapping from computations to processing elements highly depends on the input problem size and the hardware capability. Needless to say, different applications would have different optimal mappings. Therefore, we believe that any *static* mapping techniques would not be satisfactory. What we want is a *dynamic* mapping technique that can automatically adapt to the runtime environment, as we are going to propose next.

3. The Qilin Programming System

Qilin is a programming system we have recently developed for heterogeneous multiprocessors. Figure 3 shows the software architecture of Qilin. At the application level, Qilin provides an API to

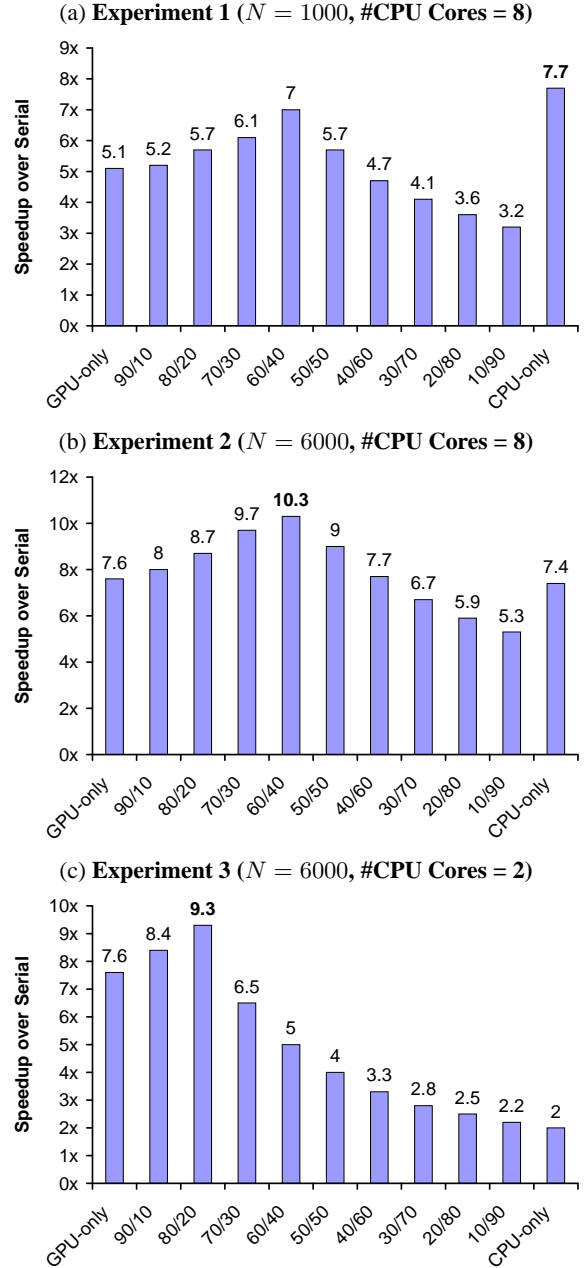


Figure 2. Matrix multiplication experiments with different input matrix sizes and number of CPU cores used. The input matrices are N by N . The notation “X/Y” on the x-axis means X% of work mapped to the GPU and Y% of work mapped to the CPU.

programmers for writing *parallelizable* operations. By explicitly expressing these computations through the API, the compiler is alleviated from the difficult job of extracting parallelism from serial code, and instead can focus on performance tuning. Similar to OpenMP, the Qilin API is built on top of C/C++ so that it can be easily adopted. But unlike standard OpenMP, where parallelization only happens on the CPU, Qilin can exploit the hardware parallelism available on both the CPU and GPU.

Beneath the API layer is the Qilin system layer, which consists of a dynamic compiler and its code cache, a number of libraries,

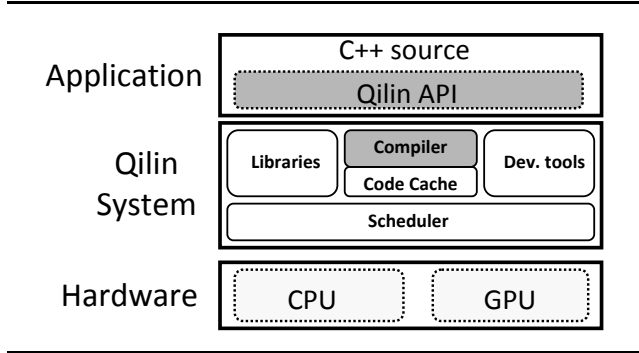


Figure 3. Qilin software architecture

a set of development tools, and a scheduler. The compiler dynamically translates the API calls into native machine codes. It also decides the near-optimal mapping from computations to processing elements using an adaptive algorithm. To reduce compilation overhead, translated codes are stored in the code cache so that they can be reused without recompilation whenever possible. Once native machine codes are available, they are scheduled to run on the CPU and/or GPU by the scheduler. Libraries include commonly used functions such as BLAS and FFT. Finally, debugging, visualization, and profiling tools can be provided to facilitate the development of Qilin programs.

Current Implementation Qilin is currently built on top of two popular C/C++ based threading APIs: The Intel Threading Building Blocks (TBB) [24] for the CPU and the Nvidia CUDA [16] for the GPU. Instead of directly generating CPU and GPU native machine codes, the Qilin compiler generates TBB and CUDA source codes from Qilin programs on the fly and then uses the system compilers to generate the final machine codes. The “Libraries” component in Figure 3 is implemented as wrapper calls to the libraries provided by the vendors: Intel MKL [8] for the CPU and Nvidia CUBLAS [15] for the GPU. For the “scheduler” component, we simply take advantage of the TBB task scheduler to schedule all CPU threads and we dedicate one CPU thread to handle all handshaking with the GPU. The “Dev. tools” component is still under development.

In the rest of this section, we will focus our discussion on the Qilin API and the Qilin compiler. We will discuss our adaptive mapping technique in Section 4.

3.1 Qilin API

Qilin defines two new types `QArray` and `QArrayList` using C++ templates. A `QArray` represents a multidimensional array of a generic type. For example, `QArray<float>` is the type for an array of floats. `QArray` is an opaque type and its actual implementation is hidden from the programmer. A `QArrayList` represents a list of `QArray` objects, and is also an opaque type.

Qilin parallel computations are performed on either `QArray`’s or `QArrayList`’s. There are two approaches to writing these computations. The first approach is called the *Stream-API approach*, where the programmer solely uses the Qilin stream API which implements common data-parallel operations including elementwise, reduction, and linear algebra operations on `QArray`’s. Our stream API is similar to those found in some GPGPU systems [2, 6, 19, 23, 28]. However, since Qilin targets for heterogeneous machines, it also allows the programmer to optionally select the processing elements. For instance, “`QArray<float> Qsum = Add(Qx, Qy, PE_SELECTOR_GPU)`” specifies that the addition of the two arrays `Qx` and `Qy` must be performed on the

```
void MySgemm(float* A, float* B, float* C, int m, int k,
            int n, float alpha, float beta)
{
    // Create Qilin arrays from normal arrays
    QArray<float> qA = QArray<float>::Create2D(m, k, A);
    QArray<float> qB = QArray<float>::Create2D(k, n, B);
    QArray<float> qC = QArray<float>::Create2D(m, n, C);

    // Invoke the Qilin version of BLAS Sgemm() on the
    // processing elements determined by the default
    // mapping scheme
    qC = BlasSgemm(qA, qB, qC, alpha, beta,
                  PE_SELECTOR_DEFAULT);

    // Convert from qC[] back to C[] and
    // this triggers the lazy evaluation
    qC.ToNormalArray(C, m*n*sizeof(float))
}
```

Figure 4. Matrix multiplication written with the Stream-API approach.

GPU. Other possible selector values are `PE_SELECTOR_CPU` for choosing the CPU and `PE_SELECTOR_DEFAULT` for choosing the default mapping scheme which can be a static one or the adaptive mapping scheme (See Section 4).

Figure 4 shows a function `MySgemm()` which uses the Qilin stream API to perform matrix multiplication. The function `QArray::Create2D()` creates a 2D `QArray` from a normal C array. Its reverse is `QArray::ToNormalArray()`, which converts a `QArray` back to a normal C array. `BlasSgemm()` is the BLAS `Sgemm()` function provided by Qilin. Since `PE_SELECTOR_DEFAULT` is used in the call, `BlasSgemm()` can be mapped to either the CPU or GPU or both, depending on the default mapping scheme.

The second approach to writing parallel computations is called the *Threading-API approach*, in which the programmer provides the parallel implementations of computations in the threading APIs on which Qilin is built (i.e., TBB on the CPU side and CUDA on the GPU side for our current implementation). A Qilin operation is then defined to glue these two implementations together. When Qilin compiles this operation, it will automatically partition computations across processing elements and eventually merge the results.

Figure 5 shows an example of using the Threading-API approach to write an image filter. First, the programmer provides a TBB implementation of the filter in `CpuFilter()` and a CUDA implementation in `GpuFilter()` (TBB and CUDA codes are omitted for clarity reasons). Since both TBB and CUDA work on normal arrays, we need to convert Qilin arrays back to normal arrays before invoking TBB and CUDA. Second, we use the Qilin function `MakeQArrayOp()` to make a new operation `myFilter` out of `CpuFilter()` and `GpuFilter()`. Third, we construct the argument list of `myFilter` from the two Qilin arrays `qSrc` and `qDst`. The keyword `QILIN_PARTITIONABLE` tells Qilin that the associated computations of both arrays can be partitioned to run on the CPU and GPU. Fourth, we call another Qilin function `ApplyQArrayOp()` to apply `myFilter` with the argument list using the default mapping scheme. The result of `ApplyQArrayOp()` is a Qilin boolean array `qSuccess` of a single element, which returns whether the operation is applied successfully. Finally, we convert `qSuccess` to a normal boolean value.

```

void CpuFilter(QArray<Pixel> qSrc, QArray<Pixel> qDst) {
    Pixel* src_cpu = qSrc.NormalArray(), dst_cpu = qDst.NormalArray();
    int height_cpu = qSrc.DimSize(0), width_cpu = qSrc.DimSize(1);

    // ... Filter implementation in TBB ...
}

void GpuFilter(QArray<Pixel> qSrc, QArray<Pixel> qDst) {
    Pixel* src_gpu = qSrc.NormalArray(), dst_gpu = qDst.NormalArray();
    int height_gpu = qSrc.DimSize(0), width_gpu = qSrc.DimSize(1);

    // ... Filter implementation in CUDA ...
}

void MyFilter(Pixel* src, Pixel* dst, int height, int width) {

    // Create Qilin arrays from normal arrays
    QArray<Pixel> qSrc = QArray<Pixel>::Create2D(height, width, src);
    QArray<Pixel> qDst = QArray<Pixel>::Create2D(height, width, dst);

    // Define myFilter as an operation that glues CpuFilter() and GpuFilter()
    QArrayOp myFilter = MakeQArrayOp("myFilter", CpuFilter, GpuFilter);

    // Build the argument list for myFilter. QILIN_PARTITIONABLE means the
    // associated computation can be partitioned to run on both CPU and GPU.
    QArrayOpArgsList argList;
    argList.Insert(qSrc, QILIN_PARTITIONABLE);
    argList.Insert(qDst, QILIN_PARTITIONABLE);

    // Apply myFilter with argList using the default mapping scheme
    QArray<BOOL> qSuccess = ApplyQArrayOp(myFilter, argList, PE_SELECTOR_DEFAULT);

    // Convert from qSuccess[] to success, and this triggers the lazy evaluation
    BOOL success;
    qSuccess.ToNormalArray(&success, sizeof(BOOL));
}

```

Figure 5. Image filter written with the Threading-API approach.

3.2 Dynamic Compilation

Qilin uses *dynamic* compilation to compile Qilin API calls into native machine codes while the program runs. The main advantage of dynamic over static compilation is able to *adapt* to changes in the runtime environment. The downside of dynamic compilation is the compilation overhead incurred at run time. However, we argue that this overhead is largely amortized in the typical Qilin usage model, where a relatively small program runs for a long period of time.

The Qilin dynamic compilation consists of the following four steps:

1. **Building Directed Acyclic Graph (DAGs) from Qilin API calls:** DAGs are built according to the *data dependencies* among `QArrays` in the API calls. These DAGs are essentially the intermediate representation which latter steps in the compilation process will operate on.
2. **Deciding the mapping from computations to processing elements:** This step either uses the programmer-specified choice at each operation (see Section 3.1) or uses the automatic adaptive mapping technique (see Section 4).
3. **Performing optimizations on DAGs:** A number of optimizations are applied to the DAGs. The most important ones are (i) operation coalescing and (ii) removal of unnecessary temporary arrays. Operation coalescing groups as many operations running on the same processing elements into a single function as possible, and thereby reducing the overhead of scheduling individual operations. It is also important to remove the allocating/deallocating and copying of temporary arrays used in the intermediate computations of `QArrays`.
4. **Code generation:** At this point, we have the optimized DAGs and their computation-to-processor mappings decided. One additional step we need to do here is to ensure that all hardware resource constraints are satisfied. The most common issue is the memory requirement of computations that are mapped to the GPU, because the amount of GPU physical memory available is relatively limited (typically less than 1GB) and it does not have virtual memory. To cope with this issue, if Qilin estimates that the GPU memory requirement of a DAG exceeds the limit, it will split the DAG into multiple smaller DAGs and run them sequentially. Once all resource constraints are taken care of, Qilin generates the native machine codes from the DAGs ac-

ording to the mappings. Qilin also automatically generates all gluing codes needed to combine the results from the CPU and the GPU.

3.2.1 Reducing Compilation Overhead

To reduce the runtime overhead, dynamic compilation is mostly done in a *lazy-evaluation* manner: When a Qilin program is executed, DAGs are built (i.e., Step 1 in the compilation process) as API calls are encountered. Nevertheless, the remaining three steps are not invoked until the computation results are really needed—when we perform a `QArray::ToNormalArray()` call to convert from a `QArray` to a normal C array. Thus, dynamically dead Qilin API calls will not cause any compilation. In addition, code generated for a particular DAG is stored in a software code cache so that if the same DAG is seen again later in the same program run, we do not have to redo Steps 2-4.

4. Adaptive Mapping

The Qilin adaptive mapping is a technique to automatically find the near-optimal mapping from computations to processing elements for the given application, problem size, and hardware configuration. To facilitate our discussion, let us introduce the following notations:

- $T_C(N)$ = The *actual* time to execute the given program of problem size N on the CPU.
- $T_G(N)$ = The *actual* time to execute the given program of problem size N on the GPU.
- $T'_C(N)$ = Qilin's projection of $T_C(N)$
- $T'_G(N)$ = Qilin's projection of $T_G(N)$

One approach to predicting $T_C(N)$ and $T_G(N)$ is to build an *analytical* performance model based on static analysis. While this approach might work well for simple programs, it is unlikely to be sufficient for more complex programs. In particular, predicting $T_C(N)$ using an analytical model is challenging with features like out-of-order execution, speculation and prefetching on modern processors. Instead, Qilin takes an *empirical* approach, as pictorialized in Figure 6 and explained below.

Qilin maintains a database that provides execution-time projection for all the programs it has ever executed. The first time that a program is run under Qilin, it is used as the *training run* (see Figure 6(a)). Suppose the input problem size for this training run is N_t . Then Qilin divides the input into two parts of size N_1 and N_2 . The N_1 part is mapped to the CPU while the N_2 part is mapped to the GPU. Within the CPU part, it further divides N_1 into m subparts $N_{1,1} \dots N_{1,m}$. Each subpart $N_{1,i}$ is run on the CPU and the execution time $T_C(N_{1,i})$ is measured. Similarly, the N_2 part is further divided into m subparts $N_{2,1} \dots N_{2,m}$ and their execution times on the GPU $T_G(N_{2,i})$'s are measured. Once all $T_C(N_{1,i})$'s and $T_G(N_{2,i})$'s are available, Qilin uses curve fitting to construct two linear equations $T'_C(N)$ and $T'_G(N)$ as projections for the actual execution times $T_C(N)$ and $T_G(N)$, respectively. That is:

$$\begin{aligned} T_C(N) &\approx T'_C(N) \\ &= a_c + b_c * N \end{aligned} \quad (1)$$

$$\begin{aligned} T_G(N) &\approx T'_G(N) \\ &= a_g + b_g * N \end{aligned} \quad (2)$$

where a_c, b_c, a_g and b_g are constant real numbers. The next time that Qilin runs the same program with a different input problem size N_r , it can use the execution-time projection stored in the database to determine the computation-to-processor mapping (see

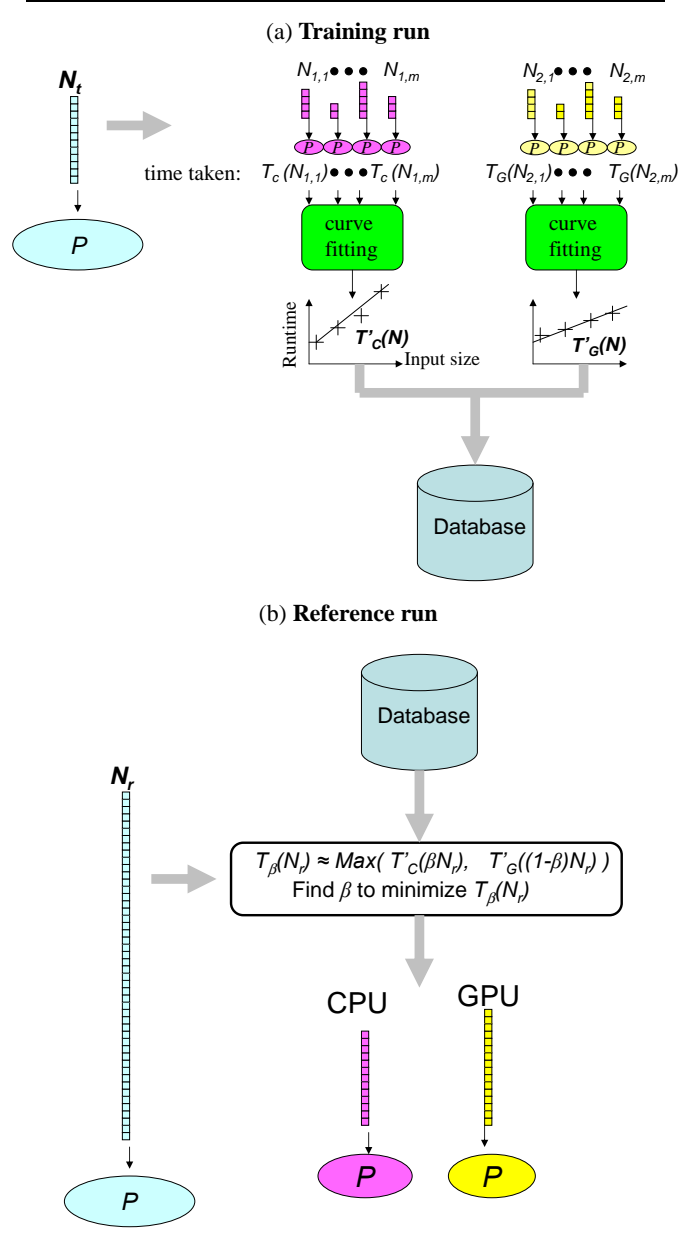


Figure 6. Qilin's adaptive mapping technique.

Figure 6(b)). Let β be the fraction of work mapped to the CPU and $T_\beta(N)$ be the actual time to execute β of work on the CPU and $(1 - \beta)$ of work on the GPU in *parallel*. Then:

$$\begin{aligned} T_\beta(N) &= \text{Max}(T_C(\beta N), T_G((1 - \beta)N)) \\ &\approx \text{Max}(T'_C(\beta N), T'_G((1 - \beta)N)) \end{aligned} \quad (3)$$

The above equations assume that running the CPU and GPU simultaneously is as fast as running them standalone. In reality, this is not the case since the CPU and GPU do contend for common resources including bus bandwidth and the need of CPU processor cycles to handshake with the GPU. So, we take these factors into account by multiplying factors into Equation (3):

$$T_\beta(N) \approx \text{Max}\left(\frac{p}{p-1} T'_C(\beta N), T'_G((1 - \beta)N)\right) \quad (4)$$

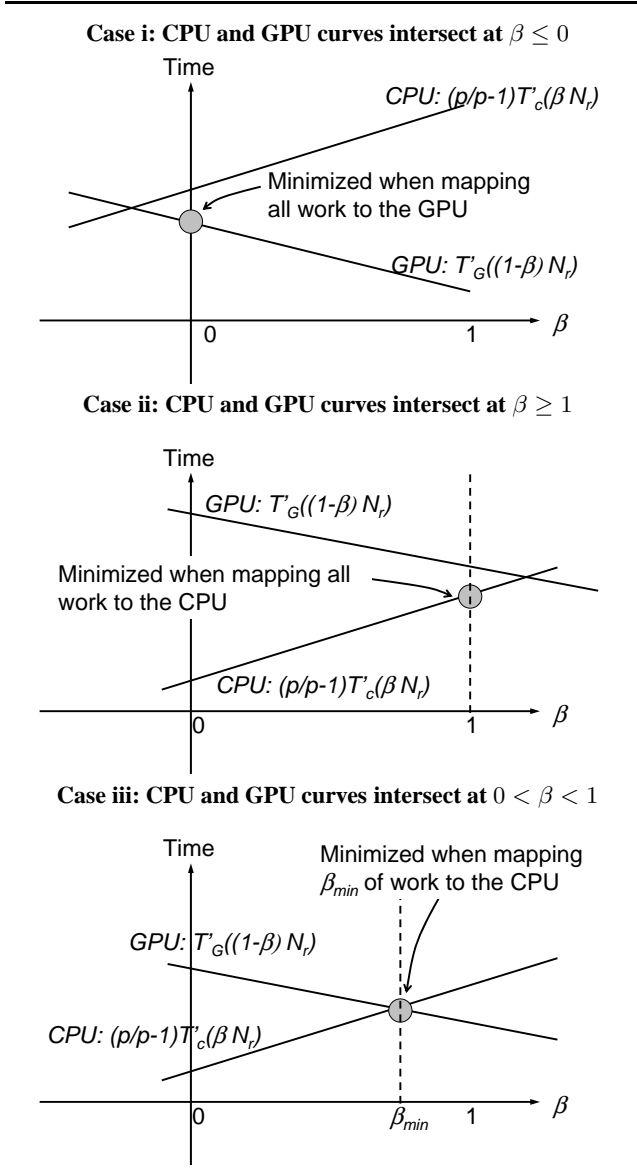


Figure 7. Three possible cases of the β in Equation (4).

where p is the number of CPU cores. Here we dedicate one CPU core to handshake with the GPU and we assume that bus bandwidth is not a major factor.

Now, we can plug the input problem size N_r into Equation (4). $T'_C(\beta N_r)$ becomes a linear equation of a single variable β and the same for $T'_G((1-\beta)N_r)$. We need to find the value of β that minimizes $\text{Max}(\frac{p}{p-1}T'_C(\beta N_r), T'_G((1-\beta)N_r))$. This is equivalent to finding the β at which $\frac{p}{p-1}T'_C(\beta N_r)$ and $T'_G((1-\beta)N_r)$ intersect. There are three possible cases, as illustrated in Figure 7. In case (i) where the intersection happens at $\beta \leq 0$, Qilin maps all work to the GPU; in case (ii) where the intersection happens at $\beta \geq 1$, Qilin maps all work to the CPU; in case (iii) where the intersection happens at $0 < \beta_{min} < 1$, Qilin maps β_{min} of work to the CPU and $1 - \beta_{min}$ of work to the GPU.

Finally, if Qilin detects any hardware changes since the last training run for a particular program was done, it will trigger a new training run for the new hardware configuration and use the results for future performance projection.

| | CPU | GPU |
|------------------|---------------------------------------|-------------------------------------|
| Architecture | Intel Core2 Quad | Nvidia 8800 GTX |
| Core Clock | 2.4 GHz | 575 MHz |
| Number of Cores | 8 cores (on two sockets) | 128 stream processors |
| Memory size | 4 GB | 768 MB |
| Memory Bandwidth | 8 GB/s | 86.4 GB/s |
| Threading API | Intel Threading Building Blocks (TBB) | Nvidia CUDA 1.1 |
| Compiler | Intel C Compiler; (ICC 10.1, "-fast") | Nvidia C Compiler (NVCC 1.1, "-O3") |
| OS | 32-bit Linux Fedora Core 6 | |

Table 1. Experimental Setup

5. Evaluation

We now present the methodology used to evaluate the effectiveness of adaptive mapping and the results.

5.1 Methodology

Our evaluation was done on a heterogeneous PC, consisting of a multicore CPU and a high-end GPU, as depicted in Table 1. Table 2 lists our benchmarks, including the ones used in the Merge work [12], the most relevant previous study, and a few other important computation kernels. We compare Qilin’s adaptive implementation against the best available CPU or GPU implementations whenever possible: for the GPU implementations, we use the ones provided in the CUDA Software Development Kit (SDK) [14] if available.

We measured the wall-clock execution time, including the time required to transfer the data between the CPU and the GPU when the GPU is used. For Qilin’s adaptive versions, we did one training run followed by a variable number of reference runs until the total execution time (including both training and reference runs) reaches one hour. We then report the average execution time per reference run (i.e., total execution time divided by the number of reference runs). This emulates the expected usage model of Qilin where a program is trained once and then used many times afterwards. In addition, since our current prototype uses source-to-source translation and invokes the system compilers to generate the final executables (see Section 3), the dynamic compilation overhead is fairly high. So we amortize it with multiple reference runs. In a production environment where we shall have a full-fledged compiler that generates code all the way from source to binary, the compilation overhead would be much less an issue.

5.2 Results

In this section, we first present results that compare adaptive mapping against manual mapping, using training set sizes identical to the reference set sizes. Second, we present results with different training input sizes. Finally, we present results with different hardware configurations.

5.2.1 Effectiveness of Adaptive Mapping

Figure 8 shows the performance comparison between automatic adaptive mapping and other mappings. The y-axis is the speedup over the serial case in logarithmic scale. The x-axis shows the benchmarks and the geometric mean. In each benchmark, there are four bars representing different mapping schemes. “CPU-always” means always scheduling all the computations to the CPU, while “GPU-always” means scheduling all the computations to the GPU. “Manual mapping” means the programmer *manually* determines the near-optimal mapping through an exhaustive search. “Adaptive mapping” means Qilin *automatically* determines the near-optimal mapping via adaptive mapping. Both “Manual mapping”

| Benchmark | Description | Input Problem Size | Serial Time | Origin |
|----------------|---|--|-------------|---------------|
| Binomial | American option pricing | 1000 options, 2048 steps | 11454 ms | CUDA SDK [14] |
| BlackScholes | European option pricing | 10000000 options | 343 ms | CUDA SDK |
| Convolve | 2D separable image convolution | 12000 x 12000 image, kernel radius = 8 | 10844 ms | CUDA SDK |
| MatrixMultiply | Dense matrix multiplication | 6000 x 6000 matrix | 37583 ms | CUDA SDK |
| Linear | Linear image filter—compute output pixel as average of a 9-pixel square | 13000 x 13000 image | 6956 ms | Merge [12] |
| Sepia | Modify RGB value to artificially age images | 13000 x 13000 image | 2307 ms | Merge |
| Smithwat | Compute scoring matrix for a pair of DNA sequences | 2000 base pairs | 26494 ms | Merge |
| Svm | Kernel from a SVM-based face classifier | 736 x 992 image | 491 ms | Merge |

Table 2. Benchmarks

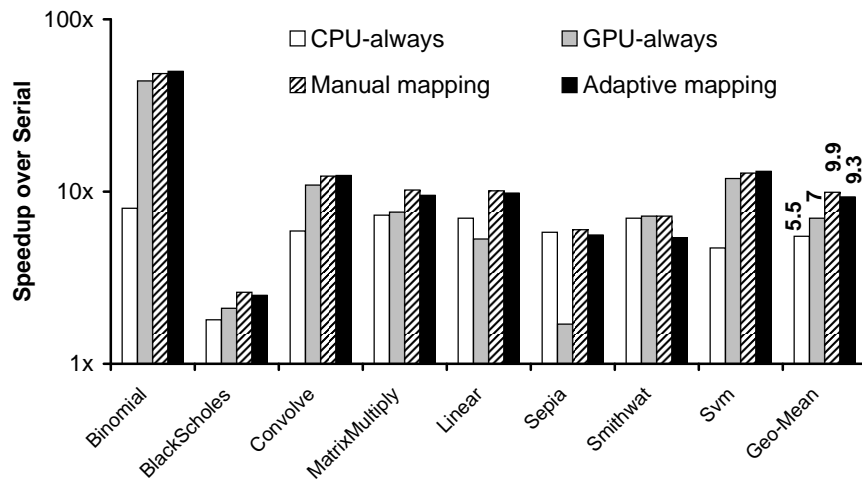


Figure 8. Performance of Adaptive Mapping (Note: The y-axis is in logarithmic scale).

and “Adaptive mapping” use training inputs that are identical to the reference inputs (we will investigate the impact of different training inputs in Section 5.2.2).

Table 3 reports the distribution of computations under the last two mapping schemes. Table 4 reports the linear equations $T'_C(N)$ and $T''_C(N)$ that Qilin constructed for each benchmark via curve fitting. Note that the equations are fairly different for different benchmarks, indicating that Qilin automatically adjusts the time predictions for different programs.

On average, adaptive mapping is 69% faster than always using the CPU and 33% faster than always using the GPU. It is within 94% of the near-optimal mapping found by the programmer via exhaustive searching. In the cases of *Binomial*, *Convolve* and *Svm*, adaptive mapping performs slightly better than manual mapping.

There are two major factors that determine the performance of adaptive mapping. The first factor is how accurate Qilin’s performance projections are (i.e., Equation (4) in Section 4). Table 3 shows that the work distributions under the two mapping schemes are similar, indicating that Qilin’s performance projections are fairly accurate. Figure 9 evaluates the accuracy of Qilin’s performance projection in more details for *Binomial*. It plots the actual and predicted execution times for both CPU and GPU with different problem sizes. As shown, the actual and predicted execution-time

| | Manual mapping | | Adaptive mapping | |
|-----------------------|----------------|-----|------------------|-------|
| | CPU | GPU | CPU | GPU |
| Binomial | 10% | 90% | 10.5% | 89.5% |
| BlackScholes | 40% | 60% | 46.5% | 53.5% |
| Convolve | 40% | 60% | 36.3% | 63.7% |
| MatrixMultiply | 40% | 60% | 45.5% | 54.5% |
| Linear | 60% | 40% | 50.8% | 49.2% |
| Sepia | 80% | 20% | 76.2% | 23.8% |
| Smithwat | 60% | 40% | 59.3% | 40.7% |
| Svm | 10% | 90% | 14.3% | 85.7% |

Table 3. Distribution of computations under the manual mapping and adaptive mapping in Figure 8.

curves are very close in both CPU and GPU cases. Similar observations are made in other benchmarks as well.

The second factor is the runtime overhead of adaptive mapping. Among our benchmarks, *Smithwat* is the only one that significantly suffers from this overhead (see the relative low performance of “Adaptive mapping” for *Smithwat* in Figure 8). This benchmark computes the scoring matrix for a pair of DNA sequences. The serial version is sketched in Figure 10(a). The two-level for-

| | CPU | | GPU | |
|----------------|---------------------------|-----------|---------------------------|-----------|
| | $T'_C(N) = a_c + b_c * N$ | | $T'_G(N) = a_g + b_g * N$ | |
| | a_c | b_c | a_g | b_g |
| Binomial | 16.21 | 1.61 | 1.20 | 0.26 |
| BlackScholes | 4.40 | 0.000015 | 0.98 | 0.000013 |
| Convolve | 6.45 | 0.18 | 4.50 | 0.10 |
| MatrixMultiply | 24.23 | 0.00017 | 96.58 | 0.00014 |
| Linear | 4.07 | 0.0000076 | 2.73 | 0.0000079 |
| Sepia | 3.80 | 0.0000028 | 4.98 | 0.0000089 |
| Smithwat | 178.85 | 2.55 | 13.71 | 3.91 |
| Svm | 3.81 | 0.0011 | 5.34 | 0.00017 |

Table 4. The time-projection equations constructed by Qilin via curve fitting (i.e., Equations (1) and (2) in Section 4).

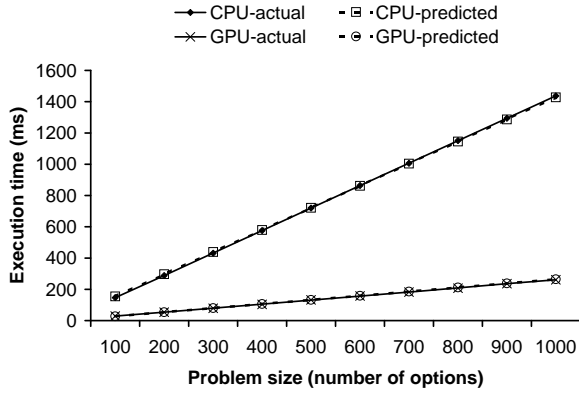


Figure 9. Accuracy of Qilin’s performance projections in Binomial.

loop considers all possible pairs of elements (one element from `seqA` and the other from `seqB`). The Qilin version is sketched in Figure 10(b). Data dependency constrains us to parallelize the inner loop instead of the outer loop. Consequently, the Qilin version has to pay the adaptation overhead in each iteration of the outer loop (i.e., the work being done in `MySmithwat()`, including converting between normal arrays and Qilin arrays and defining and applying new `QArrayOp`’s.)

5.2.2 Impact of Training Input Size

Figure 11 shows the impact of the training set size on the performance of adaptive mapping. Each benchmark uses six different training set sizes, ranging from 10% to 100% of the reference set size (i.e., the “100%” bars in Figure 11 are the same as the “Adaptive mapping” bars in Figure 8). Figure 11 shows that much of the performance benefit of adaptive mapping is preserved as long as the training set size is at least 30% of the reference set size. When the training set size is only one-tenth of the reference set size, the average adaptive-mapping speedup drops to 7.5x, but is still higher than the 7.0x or 5.5x speedups by using the GPU or CPU alone, respectively. These results provide a guideline on when Qilin should apply adaptive mapping if the actual problem sizes are significantly different from the training input sizes stored in the database.

5.2.3 Adapting to Hardware Changes

One important advantage of adaptive mapping is its capability to adjust to hardware changes. To demonstrate this advantage, we did the following two experiments.

In the first experiment, we replaced our GPU by a less powerful one (Nvidia 8800 GTX) but kept the same 8-core CPU. 8800 GTX has fewer stream processors (96) and less memory

```

(a) Serial version
void SerialSmithwat(float* score,
                   float* seqA, int startA, int lenA,
                   float* seqB, int startB, int lenB)
{
    for (int i=startA; i<lenA; i++)
        for (int j=startB; j<lenB; j++)
            ScoreOnePair(score, seqA, i, lenA,
                        seqB, j, lenB);
}

(b) Qilin version
void MySmithwat(float* score, float* seqA, int i, int lenA,
               float* seqB, int startB, int lenB) {

    QArray<float> q1 = QArray<float>::Create1D(...);
    QArray<float> q2 = QArray<float>::Create1D(...);

    QArrayOp mySmithwat = MakeQArrayOp(...);
    QArrayOpArgsList argList;
    argList.Insert(q1, ...); argList.Insert(q2, ...);

    QArray<BOOL> qSuccess = ApplyQArrayOp(mySmithwat,
                                         argList, ...);

    qSuccess.ToNormalArray(...);
}

void QilinSmithwat(float* score,
                  float* seqA, int startA, int lenA,
                  float* seqB, int startB, int lenB) {
    for (int i=startA; i<lenA; i++)
        MySmithwat(score, seqA, i, lenA, seqB, startB, lenB);
}

```

Figure 10. Explaining the high adaptation overhead in Smithwat.

(640MB) than 8800 GTX. The performance of adaptive mapping with this new hardware configuration is shown in Figure 12(a). The “GPU-always” speedup with 8800 GTX is 5.7x compared to the 7x speedup with 8800 GTX in Figure 8, or a 19% performance reduction. Adaptive mapping automatically re-distributes the computations for this change. The new work distribution is shown under the “Less powerful GPU” column in Table 5. Comparing this against the original work distribution in Table 3, Qilin has shifted more work from the GPU to the CPU. Consequently, adaptive mapping achieves a 8.2x speedup, or a 12% performance reduction compared against its 9.3x speedup in Figure 8. Note that the performance reduction in the “Adaptive mapping” case (12%) is less than that in the “GPU-always” case (19%), because Qilin is able to recover part of the loss from the CPU.

In the second experiment, we went for the other extreme, where we replaced our 8-core CPU by a 2-core CPU but kept the original GPU (8800 GTX). The performance with this new hardware configuration is shown in Figure 12(b) and the new work distribution is shown under the “Less powerful CPU” column in Table 5. With a 2-core CPU, the average speedup of “CPU-always” is down to 1.5x. Qilin decided to shift most computations to the GPU. As a result, the “Adaptive mapping” speedup in Figure 12(b) is only slightly better than the “GPU-always” speedup.

6. Related Work

Heterogeneous multiprocessors have been drawing increasing attentions from both the hardware and software research communities. On the hardware side, Kumar *et al.* [10] demonstrate the advantages of heterogeneous chip multiprocessors (CMPs) over ho-

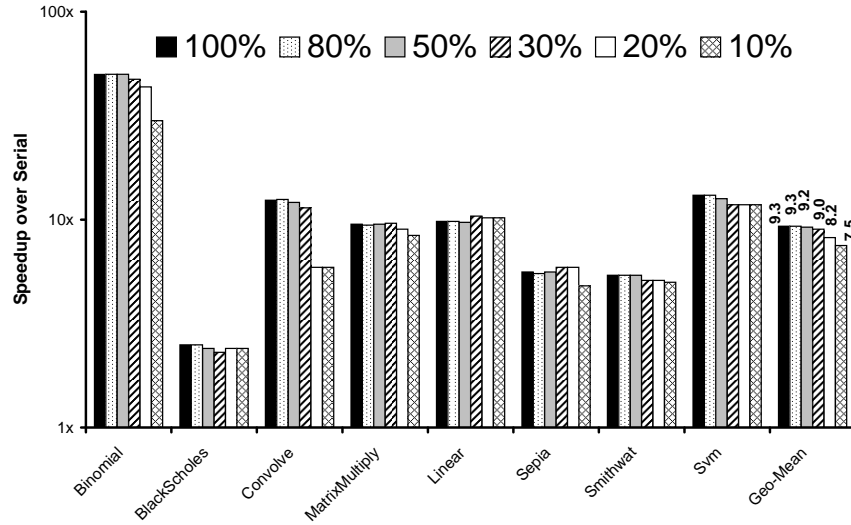


Figure 11. Impact of the training set size on adaptive mapping performance (Note: The y-axis is in logarithmic scale. The legend "X%" means the training set size is X% of the reference set size.).

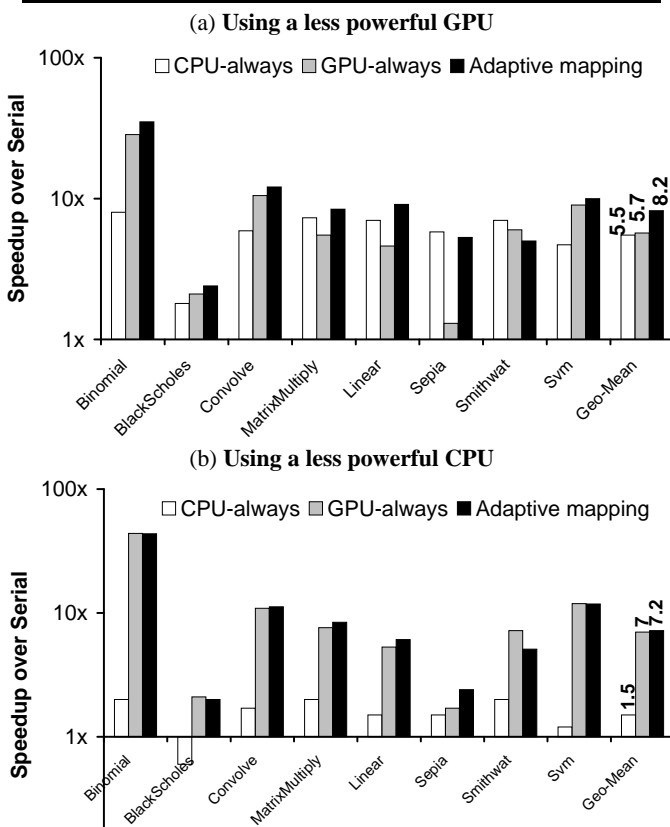


Figure 12. Performance of Adaptive Mapping with respect to hardware changes (Note: The y-axis is in logarithmic scale).

homogeneous CMPs in terms of power and throughput. They predict that once homogeneous CMPs reach a total of four cores, the benefits of heterogeneity will outweigh the benefits of additional homogeneous cores in many applications. They also classify heterogeneous multiprocessors into multi-ISA multiprocessors such as the

Cell and CPU+GPU and single-ISA multiprocessors [9] such as the upcoming Intel's Larrabee [26]. Our adaptive mapping technique is applicable to both single-ISA or multi-ISA heterogeneous multicores. Recently, Hill and Marty [7] also argue that heterogeneous ("asymmetric" in their terminology) multicore designs offer greater potential speedup than homogeneous ("symmetric") designs, provided that software challenges like the computation-to-processor mapping problem can be addressed.

On the software side, there are a number of GPGPU programming systems from both academic and industry. Most of them, including Stanford's Brook [2], Microsoft's Accelerator [28], Google's Peakstream [19], Rapidmind [23], AMD's Brook+ [1] and Nvidia's Cuda [16] target for only the GPU. In contrast, Intel's Ct [6] currently targets for the CPU only. Liao *et al.* [11] developed a compiler that generates multithreaded CPU codes from Brook-like programs. Similarly, Stratton *et al.* [27] have developed MCUDA for translating CUDA kernels to run on multicore CPUs. While Qilin shares some similarities with these systems in its stream API, it differs from them by taking advantage of both the CPU and the GPU simultaneously.

As we mentioned in Section 1, the IBM's OpenMP extension for Cell [17] and Intel's Merge work [12] are most related to Qilin as they could also map computations to both the host processor and the special processor. However, the key advantage of Qilin is that the mapping is done automatically and is adaptive to changes in the runtime environment. Most recently, OpenCL [13] is proposed as the standard API for programming GPUs. At this moment, it is unclear what kind of mechanism will be available in OpenCL to help programmers decide the computation-to-processor mapping. At the operating system level, Ghiasi *et al.* [5] proposes a scheduler that schedules memory-bound tasks to cores running at lower frequencies, thereby limiting system power while minimizing total performance loss.

Our work is also related to program *autotuning* [3, 4, 18, 21, 22, 25, 29, 31], an increasingly popular approach to producing high-quality portable code by generating many variants of a computation kernel and benchmarking each variant on the target platform. Existing autotuners largely focus on tuning the program parameters that affect the memory-hierarchy performance, such as the cache blocking factor and prefetch distance. Adaptive mapping can be viewed

| | Less powerful GPU | | Less powerful CPU | |
|-----------------------|-------------------|-------|-------------------|-------|
| | CPU | GPU | CPU | GPU |
| Binomial | 19.2% | 80.8% | 0% | 100% |
| BlackScholes | 46.3% | 53.7% | 9.9% | 90.1% |
| Convolve | 39.4% | 60.6% | 9.4% | 90.6% |
| MatrixMultiply | 53.6% | 46.4% | 11.7% | 88.3% |
| Linear | 55.3% | 44.7% | 14.5% | 85.5% |
| Sepia | 82% | 18% | 32.4% | 67.6% |
| Smithwat | 60% | 40% | 22.9% | 77.1% |
| Svm | 15% | 85% | 0% | 100% |

Table 5. Distribution of computations under adaptive mapping corresponding to the two hardware changes in Figure 12.

as an autotuning technique that tunes for the distribution of works on heterogeneous multiprocessors.

7. Conclusion

We have presented adaptive mapping, the first automatic technique that maps computations to processing elements on heterogeneous multiprocessors. We have implemented it in an experimental system called Qilin for programming CPUs and GPUs. We demonstrate that automated adaptive mapping performs close to manual mapping and can adapt to changes in input problem sizes and hardware configurations. We believe that adaptive mapping could be an important technique in the multicore software stack.

References

- [1] AMD. *AMD Stream SDK User Guide v 1.2.1-beta*, Oct 2008.
- [2] BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics* 23, 3 (2004), 777–786.
- [3] CHEN, C., CHAME, J., NELSON, Y. L., DINIZ, P., HALL, M., AND LUCAS, R. Compiler-Assisted Performance Tuning. In *Proceedings of SciDAC 2007, Journal of Physics: Conference Series* (June 2007).
- [4] FURSIN, G. G., O’BOYLE, M. F. P., AND KNIJNENBURG, P. M. W. Evaluating Iterative Compilation. In *Proceedings of the 2002 Workshop on Languages and Compilers for Parallel Computing*.
- [5] GHIASI, S., KELLER, T., AND RAWSON, F. Scheduling for Heterogeneous Processors in Server Systems. In *Proceedings of the 2nd Conference on Computing Frontiers* (May 2005), pp. 199–210.
- [6] GHULOUM, A., SMITH, T., WU, G., ZHOU, X., FANG, J., GUO, P., SO, B., RAJAGOPALAN, M., CHEN, Y., AND CHEN, B. Future-Proof Data Parallel Algorithms and Software On Intel Multi-Core Architecture. *Intel Technology Journal* 11, 4, 333–348.
- [7] HILL, M., AND MARTY, M. R. Amdahl’s Law in the Multicore Era. *IEEE Computer* (July 2008), 33–38.
- [8] INTEL. *Intel Math Kernel Library Reference Manual*, Sept 2007.
- [9] KUMAR, R., FARKAS, K. I., JOUPEI, N. P., RANGANATHAN, P., AND TULLSEN, D. Single-ISA Heterogeneous Multicore Architectures: The Potential for Processor Power Reduction. In *Proceedings of the MICRO’03* (December 2003), pp. 81–92.
- [10] KUMAR, R., TULLSEN, D., JOUPEI, N., AND RANGANATHAN, P. Heterogeneous Chip Multiprocessors. *IEEE Computer* (November 2005), 32–38.
- [11] LIAO, S.-W., DU, Z., WU, G., AND LUEH, G.-Y. Data and Computation Transformations for Brook Streaming Applications on Multiprocessors. In *Proceedings of the 4th Conference on CGO* (March 2006), pp. 196–207.
- [12] LINDERMAN, M. D., COLLINS, J. D., WANG, H., AND MENG, T. H. Merge: A Programming Model for Heterogeneous Multi-core Systems. In *Proceedings of the 2008 ASPLOS* (March 2008).
- [13] MUNSHI, A. OpenCL Parallel Computing on the GPU and CPU. In *ACM SIGGRAPH 2008* (2008).
- [14] NVIDIA. *CUDA SDK*. http://www.nvidia.com/object/cuda_get.html.
- [15] NVIDIA. *CUDA CUBLAS Reference Manual*, June 2007.
- [16] NVIDIA. *CUDA Programming Guide v 1.0*, June 2007.
- [17] O’BRIEN, K., O’BRIEN, K., SUR, Z., CHEN, T., AND ZHANG, T. Supporting OpenMP on Cell. *International Journal on Parallel Programming* 36 (2008), 289–311.
- [18] PAN, Z., AND EIGENMANN, R. PEAL—A Fast and Effective Performance Tuning System via Compiler Optimization Orchestration. *ACM Transactions on Programming Languages and Systems* 30, 3 (May 2008).
- [19] PEAKSTREAM. *Peakstream Stream Platform API C++ Programming Guide v 1.0*, May 2007.
- [20] PHAM, D., ASANO, S., BOLLIGER, M., DAY, M. M., HOFSTEE, H. P., JOHNS, C., KAHLE, J., KAMEYAMA, A., KEATY, J., MASUBUCHI, Y., RILEY, M., SHIPPY, D., STASIAK, D., SUZUOKI, M., WANG, M., WARNOCK, J., WEITZEL, S., WENDEL, D., YAMAZAKI, T., AND YAZAWA, K. The Design and Implementation of a First-Generation CELL Processor. In *IEEE International Solid-State Circuits Conference* (May 2005), pp. 49–52.
- [21] POUCHET, L.-N., BASTOUL, C., COHEN, A., AND CAVAZOS, J. Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time. In *Proceedings of the ACM SIGPLAN 08 Conference on PLDI* (June 2008).
- [22] PUSCHEL, M., MOURA, J., JOHNSON, J., PAUDA, D., VELOSO, M., SINGER, B., XIONG, J., FRANCHETTI, F., GACIC, A., VORONENKO, Y., CHEN, K., JOHNSON, R., AND RIZZOLO, N. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaption* 93, 2 (2005), 232–275.
- [23] RAPIDMIND. *Rapidmind*. <http://www.rapidmind.net>.
- [24] REINDERS, J. *Intel Threading Building Blocks*. O’Reilly, July 2007.
- [25] REN, M., PARK, J., HOUSTON, M., AIKEN, A., AND DALLY, W. J. A Tuning Framework for Software-Managed Memory Hierarchies. In *Proceedings of the 2008 International Conference on PACT*.
- [26] SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. Larrabee: A Many-Core x86 Architecture for Visual Computing. In *Proceedings of ACM SIGGRAPH 2008* (2008).
- [27] STRATTON, J. A., STONE, S. S., AND M. W. HWU, W. MCUDA: An Efficient Implementation of CUDA Kernels from Multi-Core CPUs. In *Proceedings of the 2008 Workshop on Languages and Compilers for Parallel Computing*.
- [28] TARDITI, D., PURI, S., AND OGLESBY, J. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. In *Proceedings of the 2006 ASPLOS* (October 2006).
- [29] VUDUC, R., DEMMEL, J., AND YELICK, K. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005, Journal of Physics: Conference Series* (June 2005).
- [30] WANG, P., COLLINS, J. D., CHINYA, G., JIANG, H., TIAN, X., GIRKAR, M., YANG, N., LUEH, G.-Y., AND WANG, H. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System. In *Proceedings of the ACM SIGPLAN 07 Conference on PLDI* (June 2007), pp. 156–166.
- [31] WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing* 27, 1-2 (2001), 3–35.